UNIVERSIDAD NACIONAL
SAN AGUSTIN

FACULTAD DE PRODUCCION Y SERVICIOS
ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

Informe sobre KD Tree Range Query & Clasificación de imágenes

Estructuras de Datos Avanzadas

EDUARDO ANTONIO SANCHEZ HINCHO
LUIS GUILLERMO VILLANUEVA FLORES

AREQUIPA
2019

1. Introducción

En ciencias de la computación, un Árbol kd (abreviatura de árbol k-dimensional) es una estructura de datos de particionado del espacio que organiza los puntos en un Espacio euclídeo de k dimensiones. Los árboles kd son un caso especial de los árboles BSP. Un árbol kd emplea sólo planos perpendiculares a uno de los ejes del sistema de coordenadas. Esto difiere de los árboles BSP, donde los planos pueden ser arbitrarios. Además, todos los nodos de un árbol kd, desde el nodo raíz hasta los nodos hoja, almacenan un punto. .

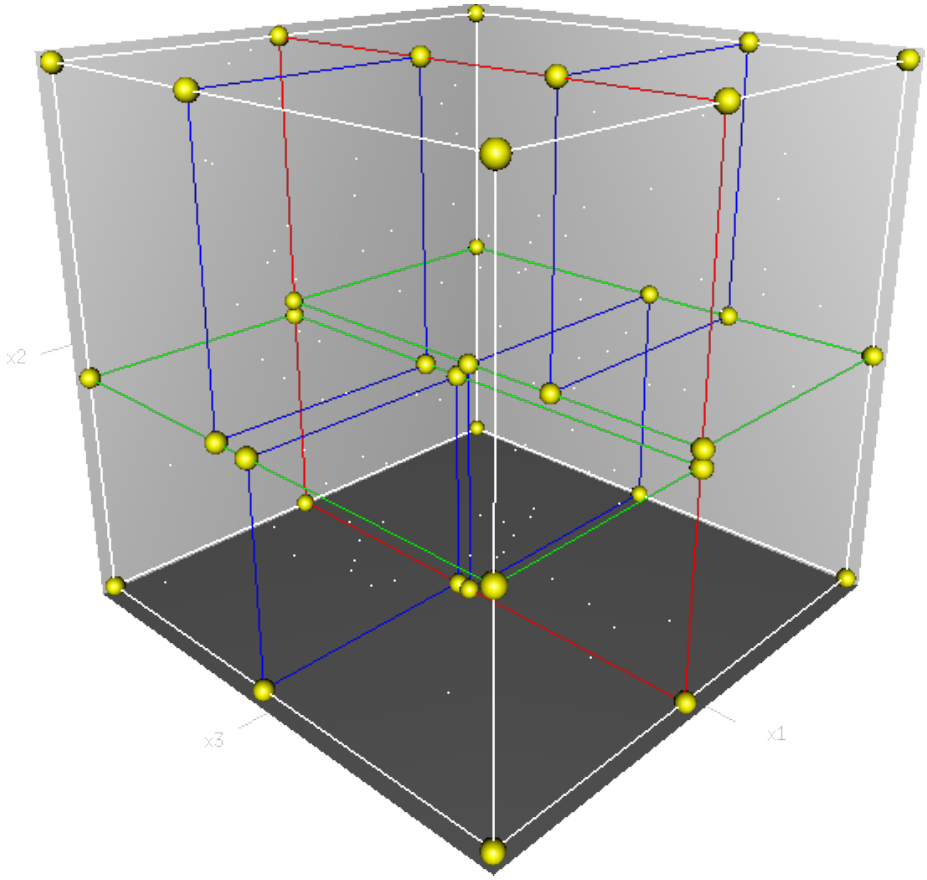

Figura 1: Representación del KD Tree en 3D

1.1. Usos del KD Tree

- Búsqueda ortogonal en un árbol kd: Usar un árbol kd para encontrar todos los puntos que se encuentran en un rectángulo determinado (o análogo de más dimensiones). Esta operación también se denomina rango de búsqueda ortogonal.
- Determinar dónde evaluar una superficie: En las regresiones locales es común evaluar la superficie contenida directamente sólo por los vértices del árbol kd e interpolar en algún punto. Este uso, reflejado en la imagen de arriba, busca asegurar que sólo se realizarán las evaluaciones directas necesarias. Como los árboles kd se .ªdaptan.ªl espacio, este método puede suministrar una excelente aproximación a las verdaderas superficies de regresión local.

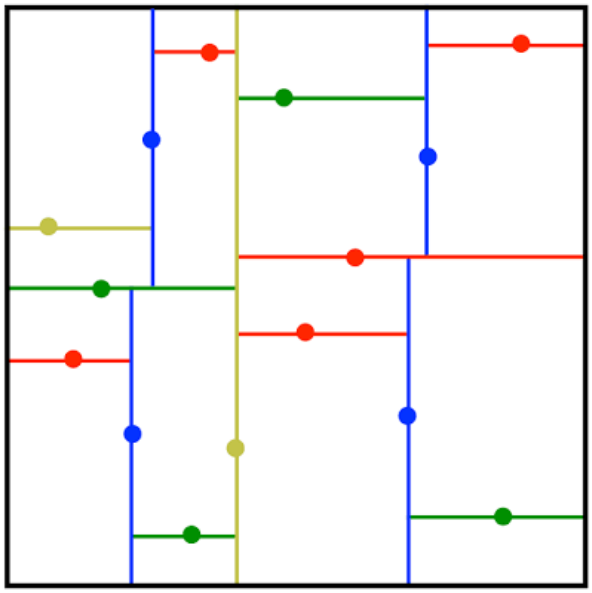

Figura 2: Representación del KD Tree en 2D

2. Range Query Circle

Para este punto implementamos nuestra función range_query_circle pero para ello en nuestra cola de prioridad definimos otro agregar, puesto que ahora se debe agregar cuando cumpla ciertas condiciones, en este caso que la distancia que le pasamos no sea mayor a nuestro radio.

```
1   agregar2(distancia,x,radio){
2     for(var i=0;i<this.lis.length;i++){
3       if(this.lis[i][1]==x)return;
4     }
5     if(distancia<radio){
6       this.lis.push([distancia,x])
7     }
8   }
```

Listing 1: Agregar para el range_query_circle

Ahora difinimos nuestra función range_query_circle que es parecida al closest_points que ya implementamos la vez pasada, pero solo con la diferencia que cambia al momento de agregar a la cola puesto por la condición que ya explique más arriba.

```
1  function range_query_circle(node,center,radio,cola,depth=0){
2    if (node == null)
3      return;
4
5    var axis = depth % k;
6    var next_branch = null; //next node brach to look for
7    var opposite_branch = null; //opposite
8    cola.agregar2(distanceSquared(center,node.point),node.point,radio
         );
9    if (point[axis] < node.point[axis]){
10     next_branch = node.left;
11     opposite_branch = node.right;
12   }else{
13     next_branch = node.right;
14     opposite_branch = node.left;
15   }
16
17   range_query_circle(next_branch,center,radio,cola,depth+1);
18   if(!cola.llena() || cola.top()>Math.abs(point[axis]-node.point[
         axis]))
19   {
20     range_query_circle(opposite_branch,center,radio,cola,depth+1)
21   }
22 }
```

Listing 2: Función range_query_circle

También definimos nuestra nueva función nearest que la llamaremos nearest2, esta se encargara de buscar los puntos cercanos al punto que le pasemos que para este caso será el centro del círculo y además que solamente estén dentro de nuestro círculo.

```
function nearest2(root,center,radio){
  var cola= new Cola(5);
  range_query_circle(root,center,radio,cola)
  return cola.lis;

}
```

Listing 3: Función nearest2

2.1. Prueba del código

Gráfico de prueba para los puntos dentro del radio 50 y que sean cercanos a [140,90]

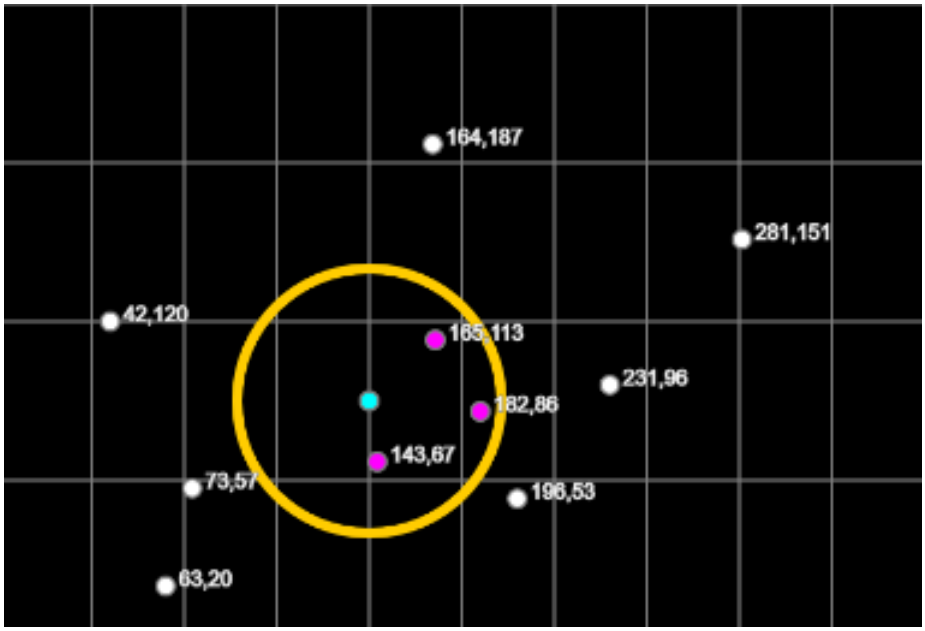

Figura 3: Gráfico de prueba

3. Range Query Rectangle

Para este punto implementamos nuestra función range_query_rectangle pero para ello en nuestra cola de prioridad definimos otro agregar, puesto que ahora se debe agregar cuando cumpla ciertas condiciones, en este caso que no sobrepase nuestro largo ni ancho de nuestro rectángulo.

```
1   agregar3(centro,x,l,a){
2     for(var i=0;i<this.lis.length;i++){
3       if(this.lis[i][1]==x)return;
4     }
5     //console.log(x)
6     if((centro[0]+(l/2)>=x[0])&&(centro[0]-(l/2)<=x[0])){
7       //console.log(x)
8       if((centro[1]+(a/2)>=x[1])&&(centro[1]-(a/2)<=x[1])){
9         this.lis.push([distanceSquared(centro,x),x])
10        //console.log(x)
11      }
12    }
13    this.lis.sort(function(a,b){
14          return a[0]-b[0];
15        });
16  }
```

Listing 4: Agregar para el range_query_rectangle

Ahora difinimos nuestra función range_query_rectangle que es parecida al closest_points que ya implementamos la vez pasada, pero solo con la diferencia que cambia al momento de agregar a la cola puesto por la condición que ya explique más arriba.

```
1  function range_query(node,center,l,a,cola,depth=0){
2    if (node == null)
3      return;
4
5    var axis = depth % k;
6    var next_branch = null; //next node brach to look for
7    var opposite_branch = null; //opposite
8    cola.agregar3(center,node.point,l,a);
9    if (point[axis] < node.point[axis]){
10     next_branch = node.left;
11     opposite_branch = node.right;
12   }else{
13     next_branch = node.right;
14     opposite_branch = node.left;
15   }
16
17   range_query(next_branch,center,l,a,cola,depth+1);
18   if(!cola.llena() || cola.top()>Math.abs(point[axis]-node.point[
       axis]))
19   {
20     range_query(opposite_branch,center,l,a,cola,depth+1)
21   }
22 }
```

Listing 5: Función range_query_rectangle

También definimos nuestra nueva función nearest que la llamaremos nearest3, esta se encargara de buscar los puntos cercanos al punto que le pasemos que para este caso será el centro del rectágulo y que no sobrepae los límites de nuestro rectángulo.

```
function nearest3(root,center,l,a){
  var cola= new Cola(1000);
  range_query(root,center,l,a,cola)
  return cola.lis;

}
```

Listing 6: Función nearest2

3.1. Prueba del código

Gráfico de prueba para los puntos dentro de largo y ancho 100 a [140,90]

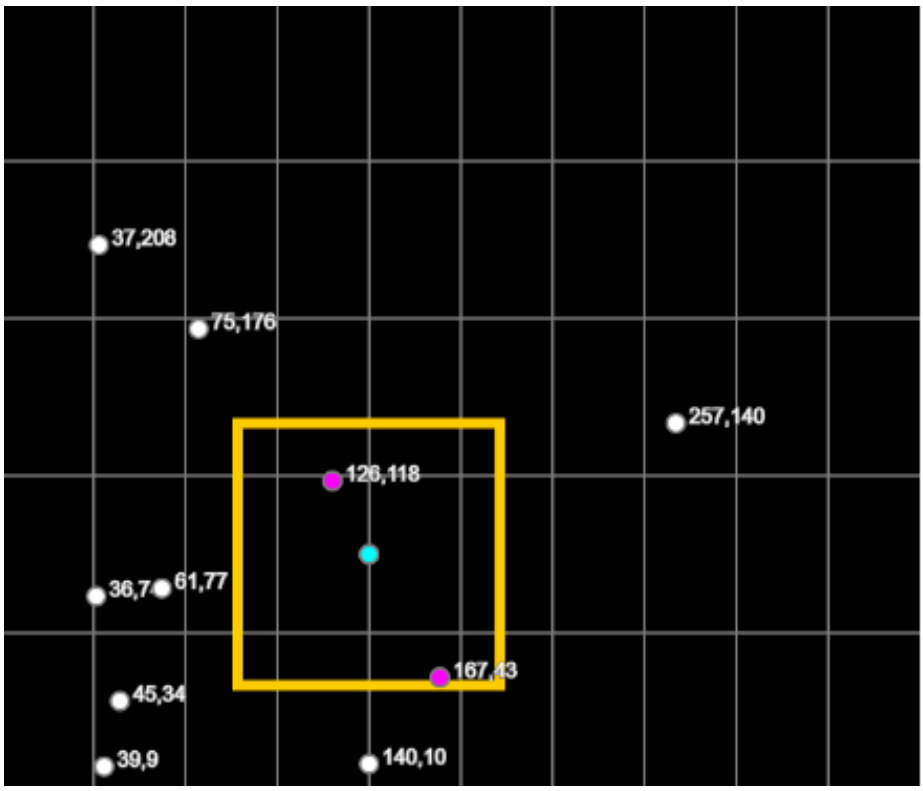

Figura 4: Gráfico de prueba

4. Clasificación de imágenes usando el KD Tree

Para este caso hemos usado el lenguaje de programación Python puesto que nos permite trabajar con imagenes más facilmente.
La idea es agarrar una imagen y la convertimos a vector y ese vector lo insertamos a nuestro KD tree que en este caso puede ser K-100 o K-200 todo depende de la dimensión de la imagen pero nosotros la reducimos a 256, sacandóle promedio a los coleres y eso recién insertamos en nuestro vector.
Código para setear la imagen a un tamaño requerido:

```
def image_to_feature_vector(image, size=(16, 16)):
    # resize the image to a fixed size, then flatten the image into
    # a list of raw pixel intensities
    return cv2.resize(image, size).flatten()
```

Listing 7: Código para setear la imagen a un tamaño requerido

Ahora definimos nuestra clase nodo y nuestra clase cola , esta última se encargara de almacenar los puntos más cercanos dependiendo a que punto querramos.

```
class Node():
    """docstring for Node"""
    def __init__(self, point):
        self.point = point
        self.axis = None
        self.left = None
        self.right = None
        self.tipo = None

class Cola():
  """docstring for Cola"""
  def __init__(self, cant):
    self.n = cant
    self.lis=[]
  def agregar(self, distancia,x):
    for i in range(len(self.lis)):
      if(self.lis[i][1]==x):
        return
    if(len(self.lis)==self.n):
      if(self.lis[self.n-1][0]>distancia):
        self.lis[self.n-1]=[distancia,x]
        self.lis.sort(key=lambda tup: tup[0])
    else:
      self.lis.append([distancia,x])
      self.lis.sort(key=lambda tup: tup[0])
  def top(self):
    return self.lis[len(self.lis)-1][0]
  def llena(self):
    if(len(self.lis)!=self.n):
      return False
    return True
```

Listing 8: Clase cola y clase nodo

Ahora definimos nuestras funciones build_kdtree, nuestra función para calcular la distancia euclidiana y por último nuestra función closest_points.

```
def build_kdtree(points, depth=0):
  if not points:
    return None
  #Para saber si dividir por el eje x o y
  axis = depth % k

  points.sort(key=lambda tup: tup.point[axis])

  median = len(points)//2

  node = points[median]
  node.axis=axis

  node.left=build_kdtree(points[:median],depth+1)
  node.right=build_kdtree(points[median+1:],depth+1)

  return node

def distanceSquared(a, b):
  distance = 0
  for i in range(k):
    distance = distance + pow((a[i]-b[i]),2)
  return math.sqrt(distance)

def closest_point(node, point, depth , cola):
  if(node == None):
    return
  axis = depth % k
  next_branch = None
  opposite_branch = None
  cola.agregar(distanceSquared(point, node.point),node)
  if(point[axis] < node.point[axis]):
    next_branch = node.left
    opposite_branch = node.right
  else:
    next_branch = node.right
    opposite_branch = node.left
  closest_point(next_branch, point, depth+1, cola)
  if(not(cola.llena()) or (cola.top() > abs(point[axis]-node.point[
      axis]))):
    closest_point(opposite_branch, point, depth+1, cola)

def nearest(root, point, count):
  cola = Cola(count)
  closest_point(root, point, 0, cola)
  return cola.lis
```

Listing 9: Funciones

Ahora implementamos la función principal , que lo que hará es tomar nuestras imagenes que estan entre loros y perros y armar nuestro KD Tree en base a un vector que ha sido transformado por las imagenes que les pasamos, ahora ese vector lo reducimos aún más puesto que le sacamos promedio para que su tamaño sea menor, luego de ello le pasamos una imagen y la convertimos a vector y sacamos los k puntos que querramos más cercanos a la imagen y contamos si los que encontro son mayor de tipo perro o mayor de tipo loro , el que reulte mayor es nuestra respuesta a la imagen insertada.

```
def main():
    #Cremos una lista con valores aleatorios para hacer el kdtree
    points=[]
    con=0
    t=0
    for i in range(1,36):
        name="perro"+str(i)+".png"
        image = cv2.imread(name)
        pixels = image_to_feature_vector(image)
        l=[]
        con=0
        t=0
        for i in range(len(pixels)):
            con=con+pixels[i]
            if(t==2):
                l.append(con/3)
                t=0
                con=0
            else:
                t=t+1


        node = Node(l)
        node.tipo="perro"
        points.append(node)

    for i in range(1,36):
        name="loro"+str(i)+".png"
        image = cv2.imread(name)
        pixels = image_to_feature_vector(image)
        l=[]
        con=0
        t=0
        for i in range(len(pixels)):
            con=con+pixels[i]
            if(t==2):
                l.append(con/3)
                t=0
                con=0
            else:
                t=t+1

        node = Node(l)
        node.tipo="loro"
        points.append(node)

    root=build_kdtree(points)
```

```
48      #image = cv2.imread("loro5.png")
49      image = cv2.imread("perro3.png")
50      pixels = image_to_feature_vector(image)
51      resul = nearest(root, pixels, 10)
52      cp=0
53      cl=0
54      for i in resul:
55        if(i[1].tipo=="perro"):
56          cp=cp+1
57        else:
58          cl=cl+1
59      if(cp>cl):
60        print("Es un perro")
61      else:
62        print("Es un loro")
63
64      print(cp,cl)
65  main()
```

Listing 10: Función principal

4.1. Carpeta de imagenes

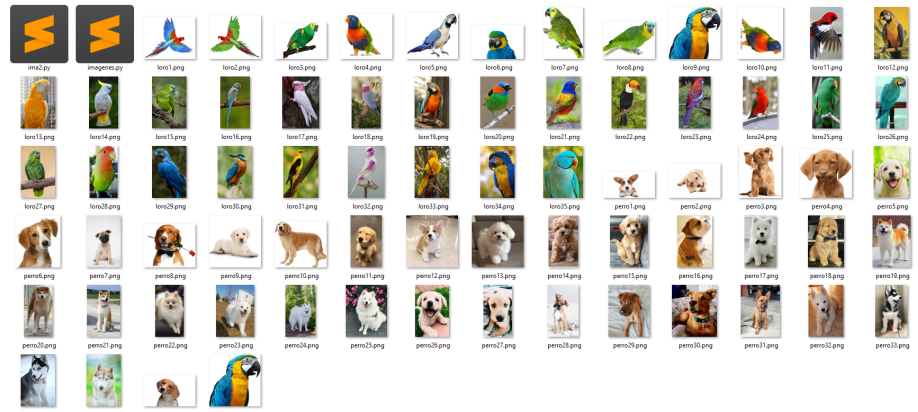

Figura 5: Carpeta de imágenes

Ahora probaremos con esta imagen llamada prueba que es un perro y nuestro código nos determinará si es un perro o loro, también nos mostrará cuantos perros y cuántos loros encontro respectivamente.

Figura 6: Imagen de prueba

Nuestro resultado sería:

```
Python 3.7.4 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: G:\knimagenes\ima2.py ==========
Es un perro
9 1
>>>
```

Figura 7: Prueba

Ahora haremos otra prueba para determinar si es loro, con esta imagen:

Figura 8: Imagen de Prueba 2

Nuestro resulado sería:

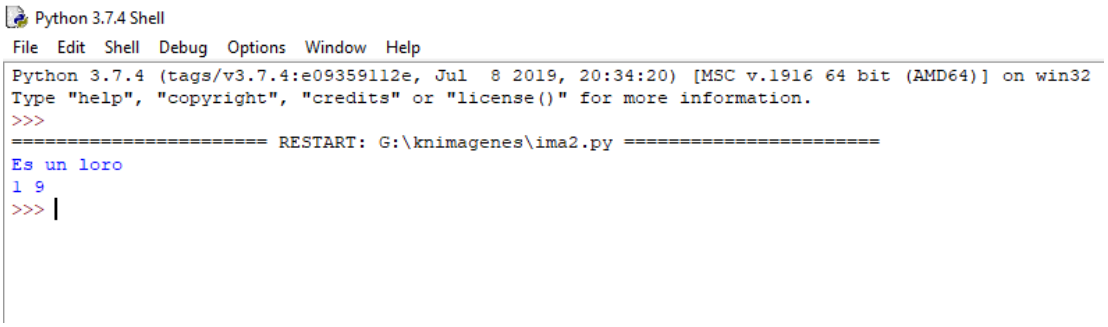

Figura 9: Prueba 2

5. Conclusiones

- Vimos que no solo sirve para almacenar puntos si no paa cualquier tipo de dato y además que nos puede ayudar a determinar si una imagen pertenece o no a un grupo.
- Vimos que de una u otra forma esto puede estar ligado a la inteligencia artificial y es muy interesante conocer todas sus aplicaciones.
- Nos dimos cuenta que al igual que en el Quadtree y Octree esta estructura nos puede ayudar mucho al momento de querer distribuir datos de una manera no uniforme, ya que pueden estar dispersos en cualquier espacio donde nosotros lo definamos y querramos que esten.
- Vimos que hay muchas maneras de hacer una búsqueda en un kd tree, pero podemos encontrarnos con algoritmos ineficientes, y eso por consecuencia nos traera problemas cuando el temaño de puntos sea mayor.
- Nos dimos cuenta que el kd tree también es bien útil en la busqueda de un punto en nuestro kd tree y eso puede tener diversas aplicaciones para proyectos futuros.
- Vimos que hay una similitud con respecto al Quadtree y Octree porque en ambos casos uno puede ser en 2D y en 3D respectivamente pero en este caso puede adaptarse para esas dimensiones y para más si en caso hubiera.
- Nos dimos cuenta que en el KDTree es mucho más fácil el almacenamiento de los datos respecto a las anteriores estructuras trabajadas.

6. Referencias

- Árbol kd. (2019). Retrieved 3 December 2019, from https://es.wikipedia.org/wiki/
- K Dimensional Tree — Set 1 (Search and Insert) - GeeksforGeeks. (2019). Retrieved 3 December 2019, from https://www.geeksforgeeks.org/k-dimensional-tree/